CHALMERS
Department of Computer Science and Engineering

2018-08-15

DAT093
Introduction to Electronic System Design
Introduction to QuestaSim

Introduction

QuestaSim is a tool for the simulation of code written in VHDL, Verilog and/or SystemC. We will focus on VHDL. The tool is just for simulation it has no options for the synthesis of hardware.

QuestaSim was created by a company called ModelTech (www.model.com). The company is nowadays a subsidiary of Mentor Graphics (www.mentor.com) and the tool is an extended version of the simulator ModelSim from the same company. The extended version can also handle PSL (Property Specification Language, a verification add-on to VHDL and Verilog) and SystemVerilog. The two tools have the same interface but with some extra menu options and features in QuestaSim. This description is written using screen dumps from QuestaSim version 10.7a. You might have ModelSim installed on your computers but you won´t notice any significant differences from QuestaSim, the same applies if you have another version of QuestaSim. There might be some small changes between different versions of the tool, but they are minor. You will also find that QuestaSim version 10,.7a is installed in the lab computers.

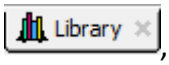
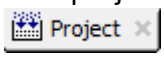
There are QuestaSim versions for both Windows and Linux. This presentation focus on the Windows version.

A version of ModelSim can be downloaded with the Xilinx design pack Vivado WebPACK, a freeware version of the Xilinx design environment Vivado.

You can also download a student version of ModelSim from the ModelTech address given above, where you will be redirected to a Mentor page. At the end of the program installation you will be urged to request a license for the software. The installation will be tied to the computer where you install it and it cannot be copied or moved. If you need to do that you must reregister and download again. These versions are somewhat limited so all the things described in this introduction cannot be done there.

Starting QuestaSim

To start QuestaSim double click on the QuestaSim icon  on the PC desktop or start it from the Start menu.

When QuestaSim open it will show a GUI with a couple of sub windows, *Figure 1*. One of these windows is the dominant Workspace window which from the beginning is filled by the Library tab  , where you can see all the standard libraries that can be used. When you have created a project a work library for the project will be added and there will also be a Project tab  where all the files that you have added to your project will be seen, *Figure 2*. When we start a simulation, there will be yet another tab.

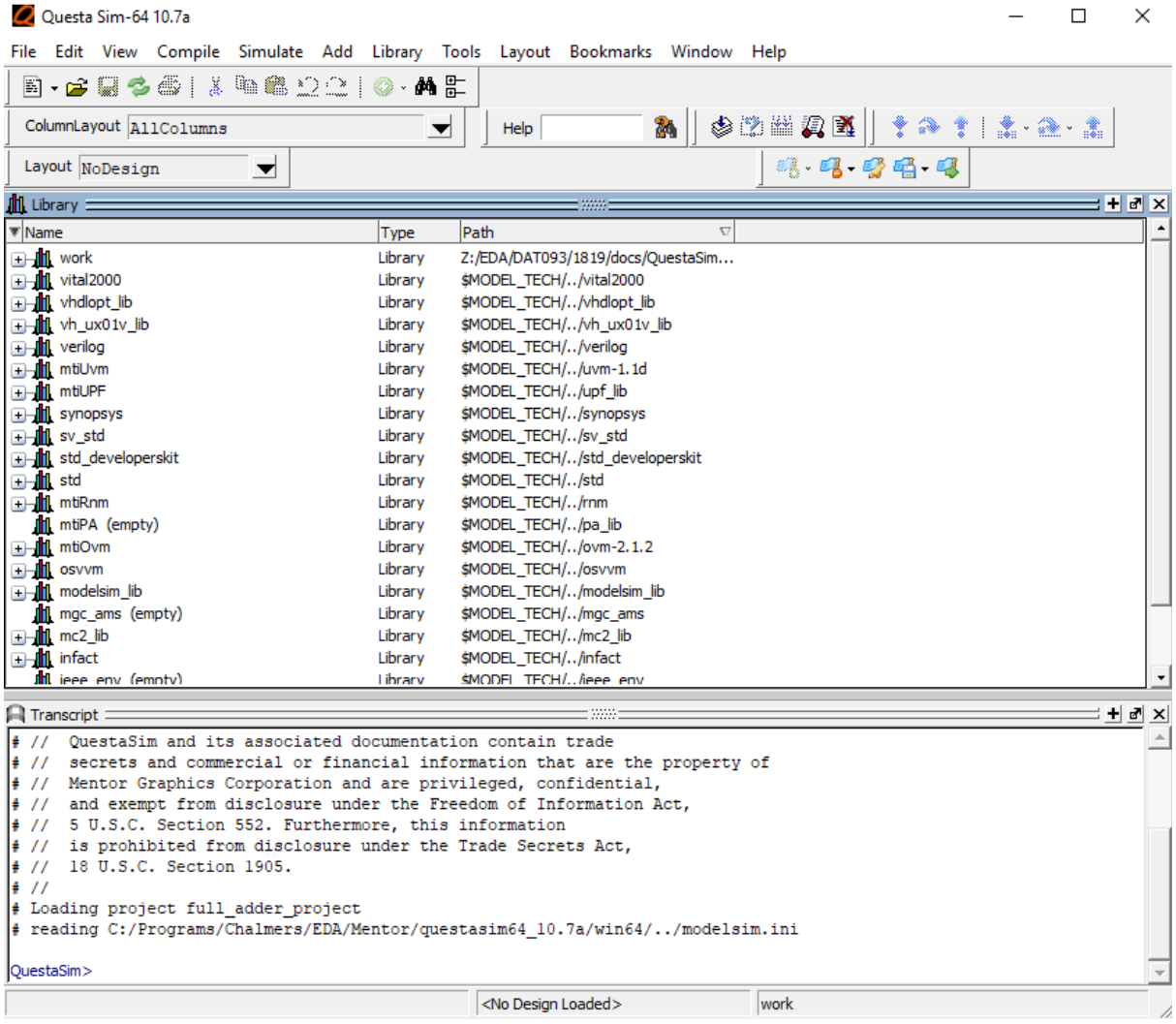


Figure 1 QuestaSim Start up GUI

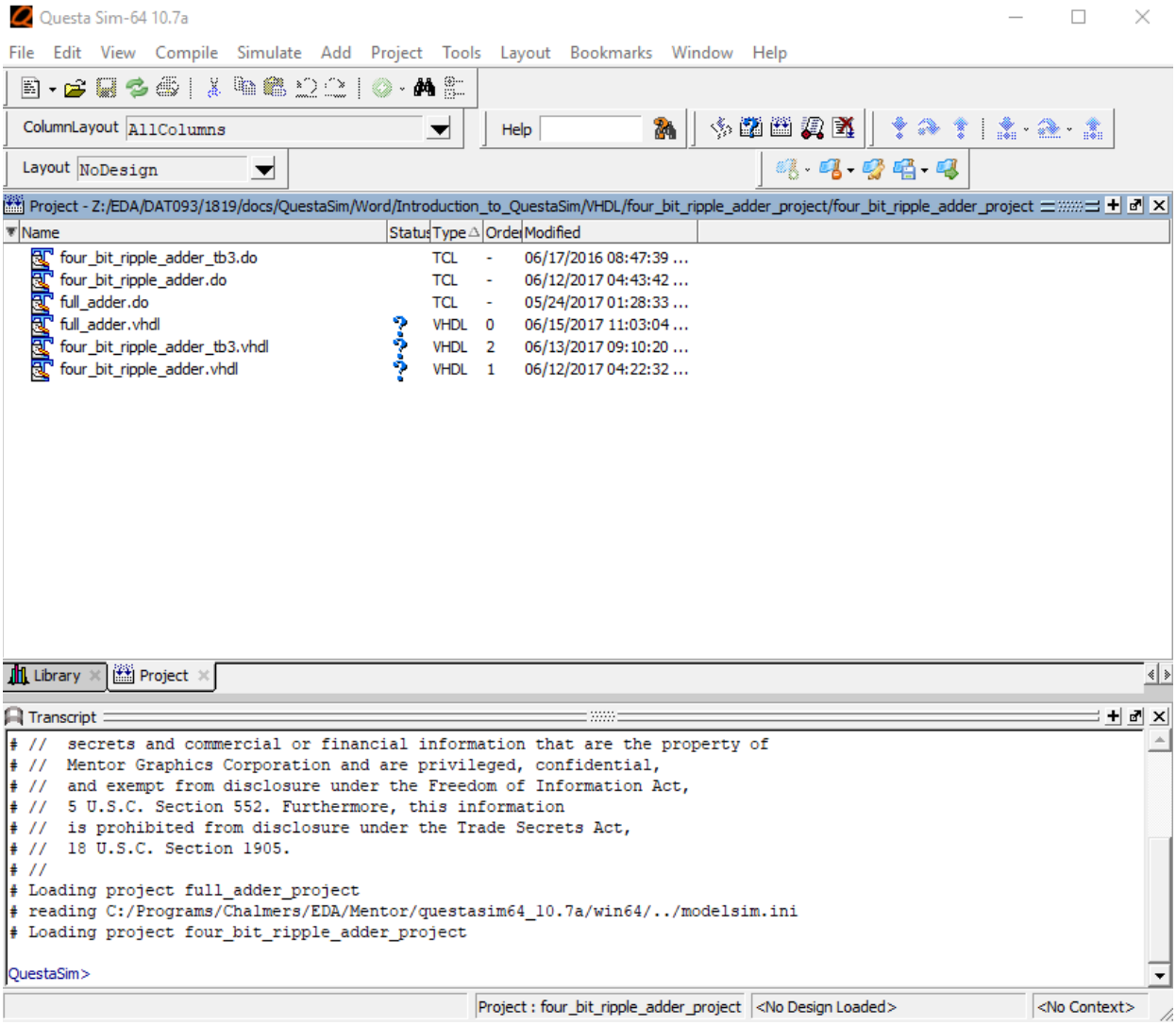
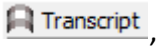


Figure 2 GUI with Project and files

Your GUI might look somewhat different from the one in *Figure 1 and 2*. The reason is that the GUI is highly configurable. A number of the tools in the top gray area can be moved around and added or removed by right clicking on the gray background and activating or de-activating options.

In *Figure 2* you can see the design files for a four-bit ripple adder with a 1-bit full adder as a component. Test bench and do file for the design are added. We´ll get back to these files soon.

From the beginning, you will also see a Transcript window  , a text-based shell where you give commands to and get messages back from the compiler and simulator.

These and all other windows can be undocked from the GUI into resizable floating windows using the  icon in the upper right corner of each window. The windows can be docked back into the GUI using the  icon that will show when the window is floating.

When you have been running a project, the environment will be saved automatically and the next time you start QuestaSim the old project will be opened again.

Creating a project

The first thing to do when you begin a design is to create a new project.

You create a new project from the File menu by selecting

File -> New -> Project…

This will give a popup window where you have a browser to select the location of the project and a line to give the project a name, *Figure 3*. Here we use a 4-bit ripple adder with test bench as an example and have given the project the name `four_bit_ripple_adder_project` and placed it in a folder with the same name. The location of the project is set in the Project Location area of the popup window, either by typing in the search path or simpler by using the Browse… button which will give a new popup window, *Figure 4*, where you can navigate to the place where you want to place the project.

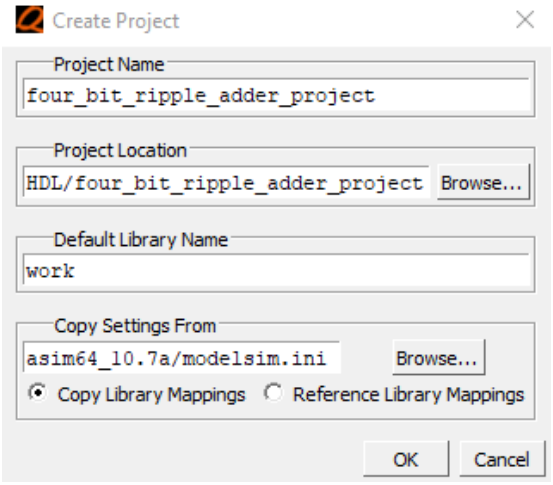


Figure 3 Project creation window

The project creation as such will not create any new folder for the project but it is a good idea to create a new folder for each project, so you can keep track of your projects and of the files within the projects.

You can do this by adding a new folder at the end of the Project Location search path line in the Create Project

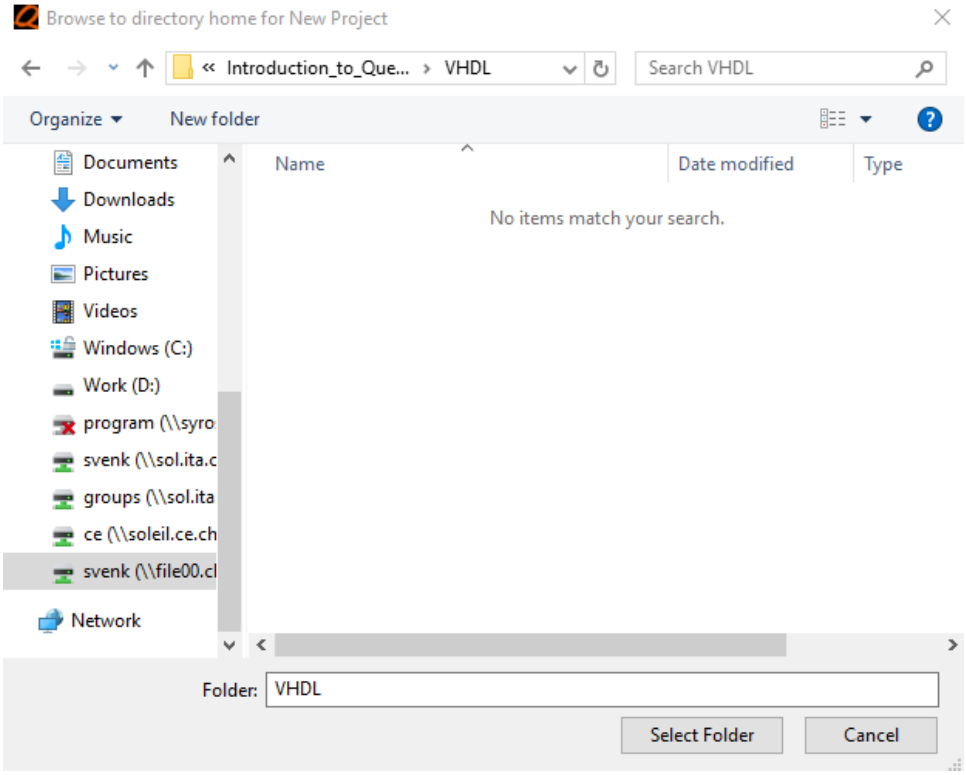


Figure 4 Folder browser

window, as we´ve done in *Figure 3*. You will then be asked if you want to create the folder. You can also add a new folder by clicking on the New folder button in the Browse to directory home for New Project window.

As said you must set a name for the new project in the Create Project window on the Project Name line and our recommendation is that the name of the project is the same as the name of the folder where the project is placed, so here we have called the project `four_bit_ripple_adder_project` and we have also added a folder with the same name to hold the project. The easiest way to set the project name is to copy the folder name from the Folder line in the Browse for Folder window, *Figure 4*, and paste it into the Project Name line in the Create Project window, *Figure 3*.
The Default Library Name input box will give the name of the subdirectory where your compiled code will be placed (work). Leave this and the other settings as they are.

Adding files to the project

When you click OK, after giving the project name, you will get a new popup window, Add items to the Project, where you can create new files or add existing files to the project, *Figure 5*.

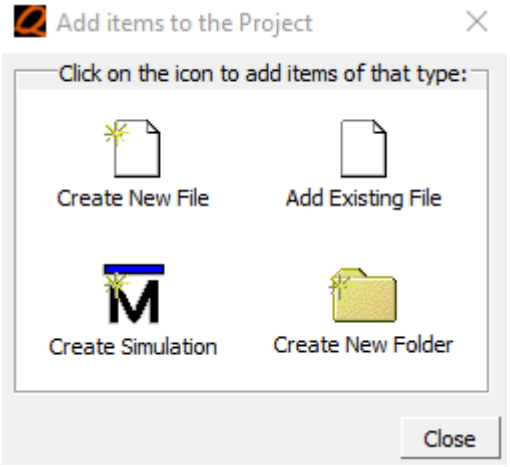


Figure 5 Add items window

The popup window you get when you click on Create New File 

, *Figure 6*, has the heading Create Project File but this does not mean the file that is defining the project, the heading should really read Create File Within the Project. When you create a new file, you are supposed to choose the type of file, for example VHDL file, from the menu Add file as type. Selecting a file type will not create any file template or anything. It will just create an empty file with the expected file ending. The created file will automatically be added to the project.

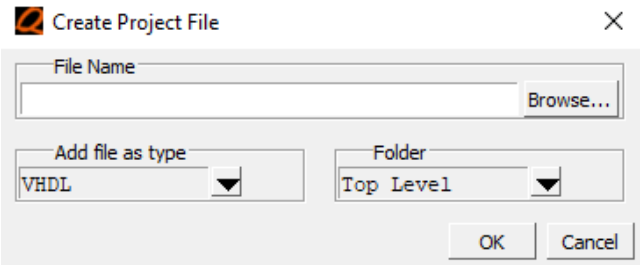


Figure 6 Create Project File window

Creating a do file, the simulation script file, is not a possibility here so you will have to create the file separately and then add it to the project. You can create the file from

File -> New -> Source -> <file type>

where File Type gives the option to set the file type.

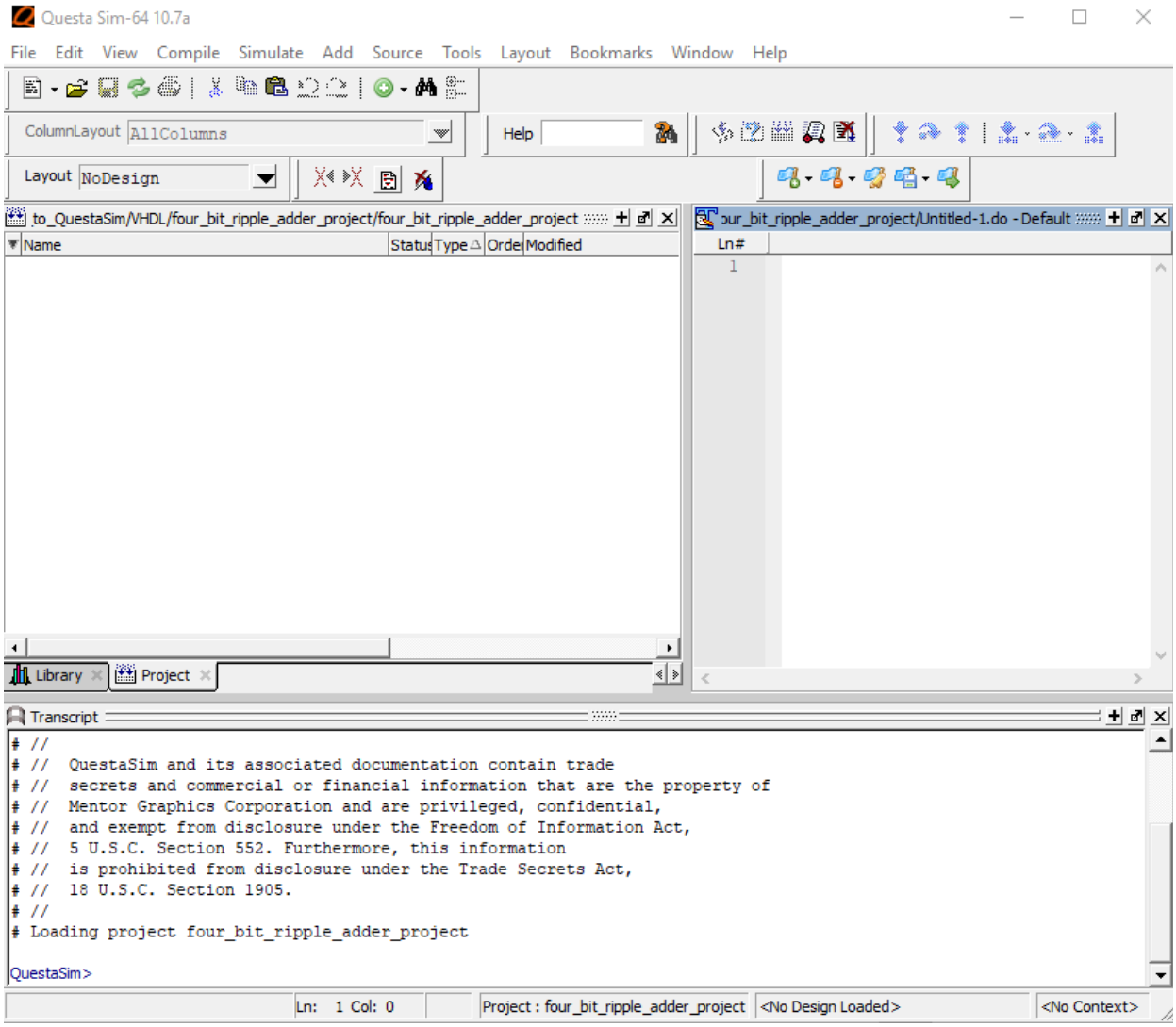


Figure 7 Adding a new do file that shows in the editor

The file will by default have the name `Untitled-#.<file_type>`.

When you create the file, it will be opened in an editor within the GUI, *Figure 7*.

When you save the file, you will be asked for a more proper file name. The `.<file_type>` ending will not be automatically added to your selected file name, so you will have to add it yourself when you give the file name.

Notice that creating the file will not automatically add the new file to the project so you will have to do that manually afterwards by right clicking in white area of the Project window and select the command

Add to Project -> Existing File…

and you get the popup menu in *Figure 7*.

You can also create a new file in the project by right clicking in the white area of the Project tab and select

Add to Project -> New File…

And you get the popup window from *Figure 6*. Doing it this way will not only create a new file but also add it to the project.

In the Add items to the Project window, *Figure 5*, you can also create new folders by clicking  and you get the dialogue in *Figure 8*. This is not of that much use since you have already created a folder for the project, so we just leave that.

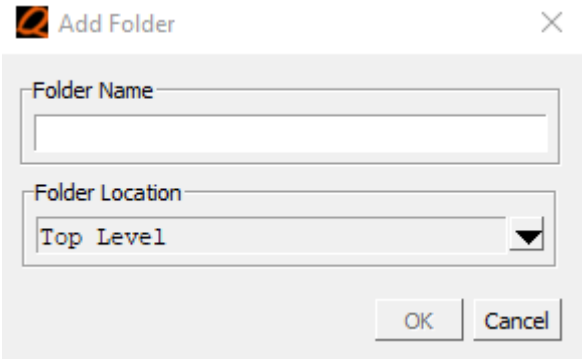


Figure 8 Create Project File window

You can also add existing files to the project using the button  . A popup window opens where you can navigate to the file, *Figure 9*.

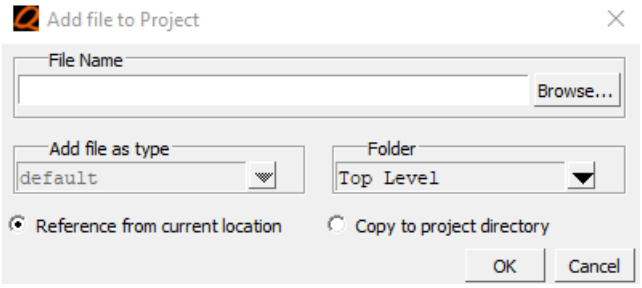
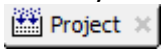


Figure 9 Add Existing File window

In *Figure 9* you have the option of keeping the file where it is or creating a copy in the project folder. In the last case, the original file will not be changed by edits within the project.

The files you add to the project do not have to be in placed the project folder.

In many cases, it simplifies things if you keep the files in the project folder but at the same time if you are using designs from another project it might be best to leave the files where they are, so you don´t end up with several versions of the same file. If you copy the file to the project and then have multiple file versions it´s hard to keep track of what´s the current version. Just beware that if you include a file from an earlier project and don´t do a copy then if you edit the file then the earlier project will also be affected.

When you have finished creating your project and get back to the GUI a new Project tab  has been added to the Workspace window containing the new or added files, as we saw in *Figure 2*. At the same time the project file and the newly created files are added to the file folder on the hard drive together with the subfolder work that will contain your compiled design files. You can create files later on when you have already created the project by using the menu command

File -> New -> Source -> <File type>

As was mentioned above.
You can also create a new file in the project by right clicking in the white area of the Project tab and select

Add to Project -> New File…

and you get the popup window from *Figure 6*. Doing it this way will not only create a new file but also add it to the project.
A file that is added to the project doesn´t have to be in the project folder. The tool will keep track of the search path. This means that if you start moving files around using a Windows file browser outside of QuestaSim the project will not find the files any more and you should remove them from the project by selecting the file, right click and select Remove from Project and add them back to the project from their new location.

Opening a project

Once you have created a project and closed it you can open it again with the menu choice

File -> Open…

which will give you a file browser where you can navigate to and choose the project file you would like to open, *Figure 9*.

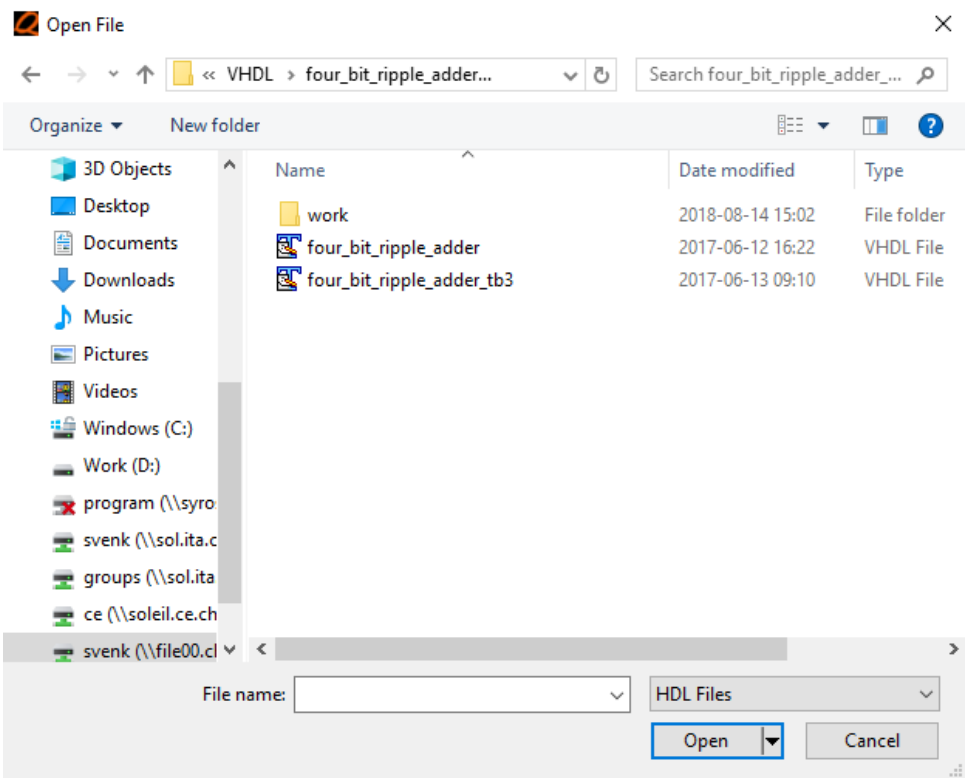


Figure 9 Open File br

The project files are not visible in the browser by default, so you will have to change the file filter from HDL Files to Project Files to see them. The project files have the ending `.mpf`.

By default, the script files, the `do` files, that we will talk about later on, are not visible in the browser either so you have to change the file filter for them to be shown. There is no filter

for selecting this kind of file, so you will have to select 

for this. The do files should have the ending .do.

Editing files

The next phase is to write or edit your VHDL code. You open the new or added VHDL file by double clicking on it in the Project tab of the Workspace window.

This will open a simple text editor which supports the common Windows shortcuts for copy, paste and so on. The editor will show line numbers and will color code the text with different colors for VHDL commands, signal types, values and others.

There is a bug in the program that has the effect that sometimes the file will not open up in the built-in editor but in an external editor like Notepad or TextPad. There is a drawback with this since these editors don´t color code VHDL. If files have started to be shown in the external editor, they will do that from that on. Support at Mentor have not managed to sort this out. One solution would be to add an external editor that can handle VHDL color coding, like Notepad++. There is a way to make the files show up in the internal editor, we´ll get back to that.

This may also have the result that you have versions of the same file both in the GUI editor and the external editor so be aware of what file you are editing.

In many cases the best way to start a new file is to include or copy a file from an old project and then do the required changes. In this way, you get the basic file structure to start with. Please remember that if you include the file any edits will also affect the functionality in the project from where the file is included since it´s the same file. It´s best to save the file under a new name.

Compiling files

Before you can simulate the design the file(s) must be compiled. You do this from the Compile menu or by right clicking in the Project tab which will open up a menu. Here you can choose to compile all design files or if you right click on a design file you can choose to only compile that file or you can compile only the files that are out of date, that is files that have been changed since the last compilation.

A common mistake is to forget to save the current file before compilation. Then the old, saved file will be compiled. You can easily see if a file is saved by looking at the save file icon  in the GUI when the file in question is active. If the icon is dimmed  then the file is saved, if the icon is sharp  then the file isn´t saved.

As you can see in *Figure 2* the VHDL files in the Project tab shows a question mark  in their Status column. This indicates that the files haven´t been compiled since they were last edited.

Before you compile all files, you can use the same menu to set the order in which the files will be compiled by selecting compile/Compile Order…, and you get *Figure 10* where you select a file and move it up or down in the order by using the arrows. This may be of use if you have a hierarchical design because then the lower level code, for example the components, have to be compiled first, before they are used in the compilation of the higher-level design. A simple way to fix this anyway is to run the compilation twice. By doing that the components that are needed at the higher level have been compiled in the first compilation round and will be available for the second round.

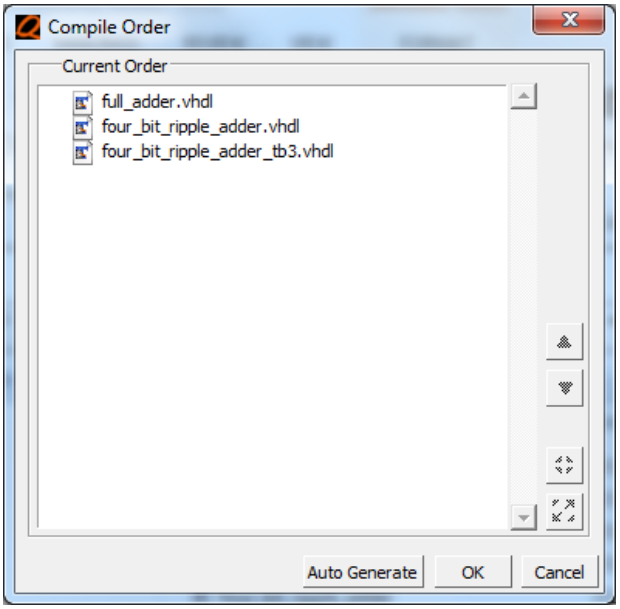


Figure 10 Setting the compile order

If the compilations succeed we get *Figure 11*.

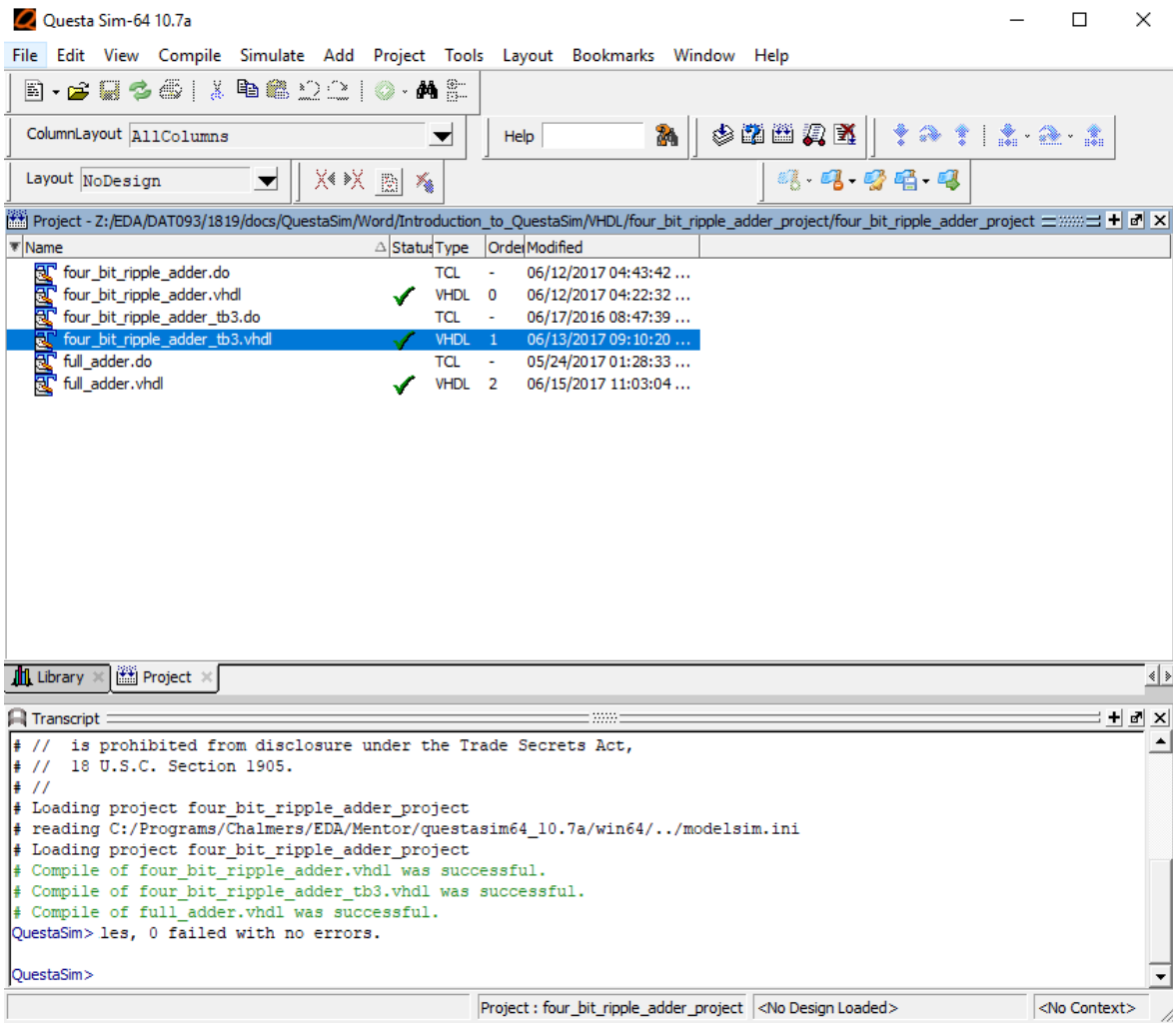


Figure 11 Successful compilation

You will see the text

```
# Compile of full_adder.vhdl was successful.
# Compile of four_bit_ripple_adder.vhdl was successful.
# Compile of four_bit_ripple_adder_tb3.vhdl was successful.
```

in green in the Transcript window and the question marks in the Status column have changed to OK signs ✓ indicating that the compilation succeeded.

In most cases your code writing will not be correct the first time so there will be some errors in your first compilation and you will have to do some corrections to successfully get the code through the compiler, *Figure 12*.

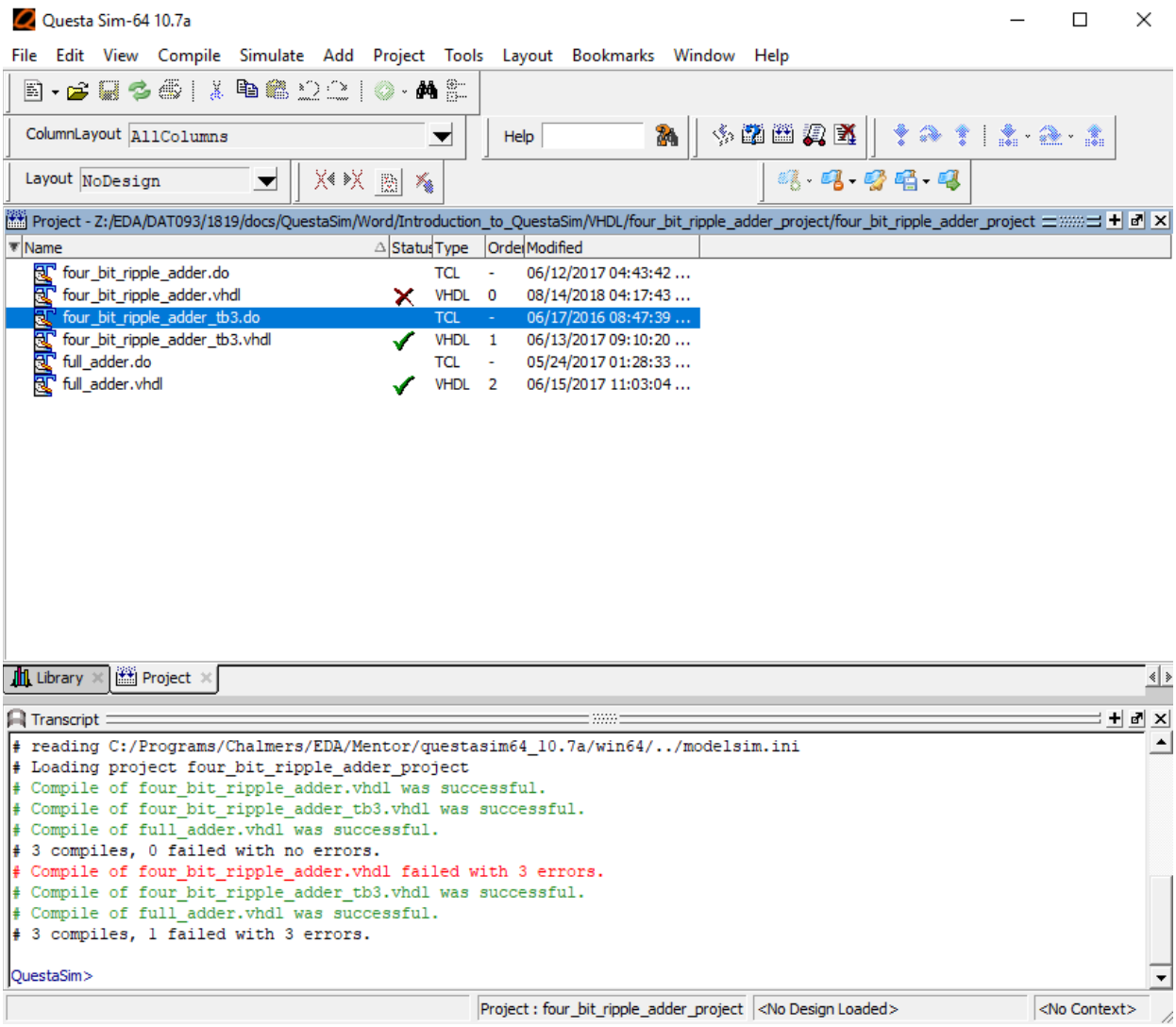


Figure 12 Compilation with error

Errors will give a red line in the Transcript window

```
# Compile of four_bit_ripple_adder.vhdl failed with 3 errors.
```

And in the Status column you get a red cross  at the file with the error(s). In *Figure 12* we can see that there is an error in the file `four_bit_ripple_adder.vhdl`.

If you double click on the red error line in the Transcript window a popup window will show the line(s) in the code where the error(s) occurred accompanied by an explanation that will hopefully assist you in correcting the error(s), *Figure 13*. Some messages are a bit cryptic, but you will by experience learn what they mean.

If you double click on one of the line(s) describing the error(s) the offending line in the code will be highlighted, *Figure 14*. Start from the first error message. For some mysterious reason the built-in editor will show when you do this although you may have been forced to use an external editor before. Note that this could have the result that the same file is open in both the internal and external editor, so you must make sure to edit the correct version.

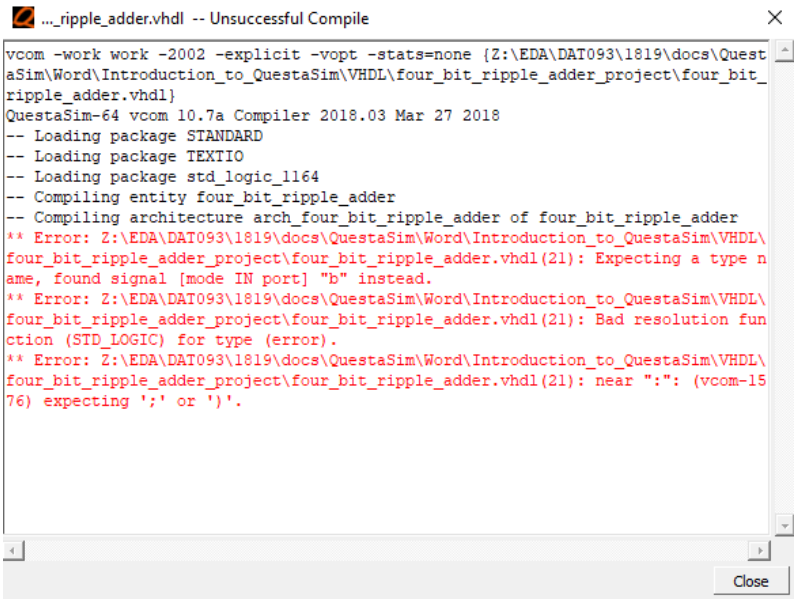


Figure 13 Error window

In many cases the highlighted line is not the one with the actual error but the error is on the line above or some lines above the highlighted line. The reason for this is that the compiler doesn´t notice the error until it finds that the next line is incorrect because of the earlier error.

In the example in *Figure 13* the actual error is a missing semicolon on line 20 although the compiler reported an error on line 21 and as we saw in *Figure 13* this resulted in three errors and these will all go away when we add the missing semicolon.

Don´t be discouraged if you get a lot of error lines in *Figure 13*. In many cases a small error can lead to a number of other errors and after correcting this small error a lot or all of the error messages might disappear but correcting an error can also give several new errors since after the correction

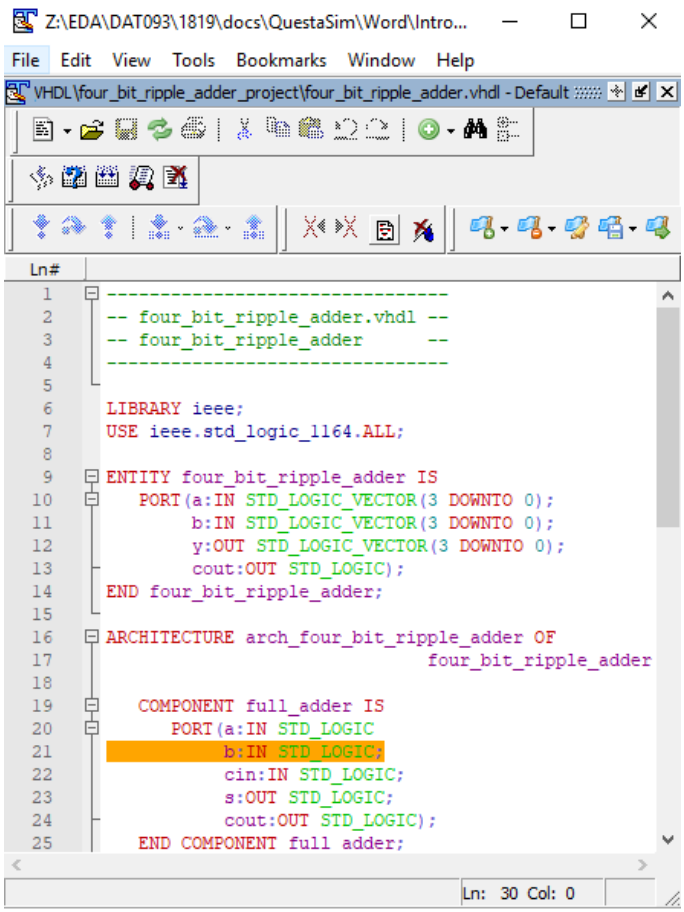


Figure 14 Code with highlighted error

the compiler can continue into parts of the code that wasn´t analyzed before.

Work your way through the errors by starting with the first one, recompile and if necessary move on to the next error.

When you have corrected all the errors and the compilation is successful you are ready for simulation.

Simulating the design

You start the simulation with the menu choice

Simulate -> Start Simulation

and this will give a popup window, *Figure 15*, where you choose what design to simulate, you can simulate the test bench, the top-level design or a component. The simulation sources are in the work folder but default this folder isn´t open so you must open it. What you see in the work folder are the entities from your design files.

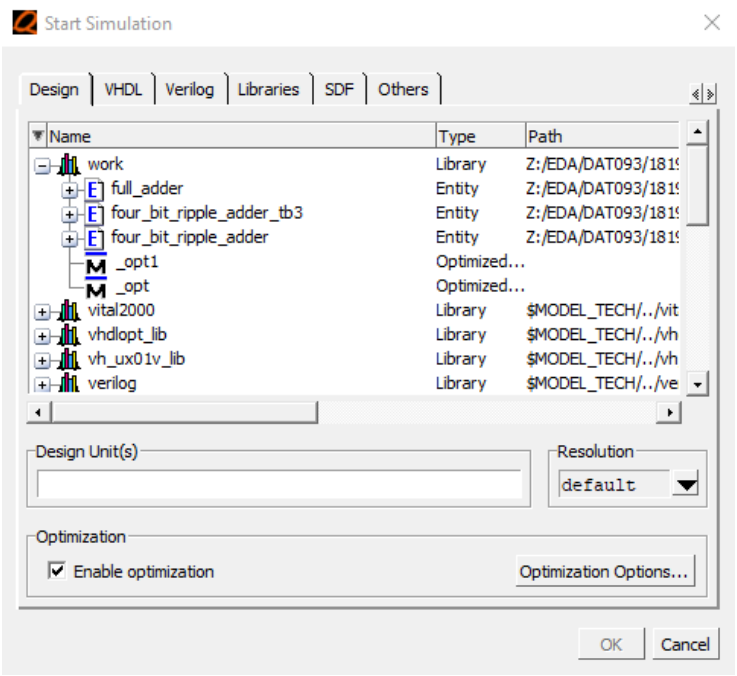


Figure 15 Start simulation window

You can also start the simulator from the Transcript window using the command

```
vsim <name of top entity>
```

In the Start Simulation window, there is also an important option named Optimization. By default, Enable optimization is checked and optimization is activated and this means that only the top-level signals and ports are certain to be visible since the optimization might have resulted in the compiler doing a redesign that removed or changed the lower level signals. In this case, you will not be able to see variables within processes either since they might be optimized away. The optimization will also remove all signals that are not contributing to any outport signal.

When we debug the design, it is often very practical to see variables and lower level signals. To do this we would like to disable optimization. There is a bug, so it will not work to just unchecking the Enable optimization option.

We can avoid optimization and keep visibility of the design components if we click on the Optimization Options... button in the Start simulation window before we select the source to simulate. You can only click on this button if Enable optimization is checked.

The window in *Figure 16* will show and here we can change the active choice from No design object visibility to Apply full visibility to all modules (full debug mode). There are also some other tabs in this window where you can fine-tune the level of optimization, but we skip that.
In *Figure 15* we can see that the work folder contains not just the design modules but also something called `_opt`. This is the result of an earlier simulation that we ran with optimization.

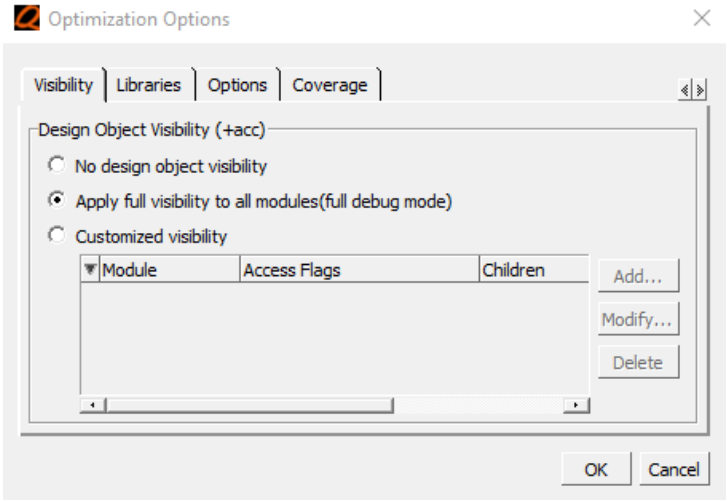
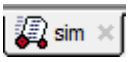


Figure 16 Optimization Options window

Another way to start the simulation is to right click on the file you want to simulate in the work folder of the Library tab and then select to simulate without optimization or with one of the optimization levels available. To run without optimization, you must first activate the window in *Figure 16* and activate Apply full visibility to all modules (full debug mode).

When you have selected the design to simulate the GUI will change its appearance, *Figure 18*, a Simulation tab 

has been added and we go into simulation mode.

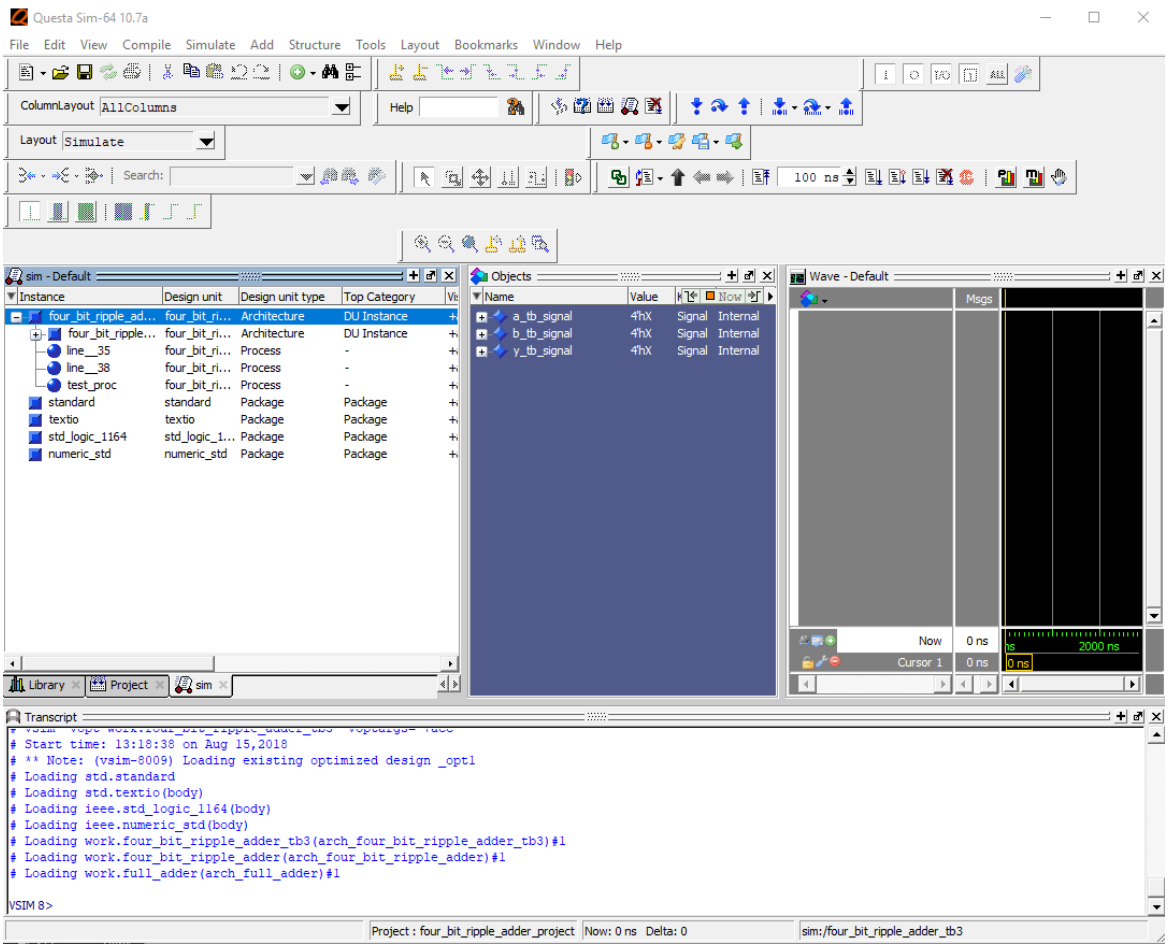


Figure 18 GUI in simulation mode

In *Figure 18* you can see the simulation of a test bench for a 4-bit ripple adder with four full adders as components.

We get a new window, the Objects window, *Figure 19*.

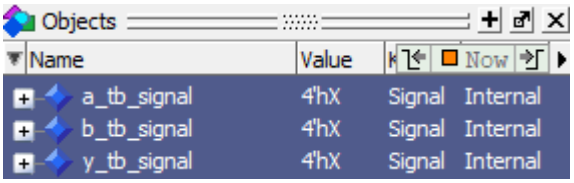


Figure 19 Objects window

In the Objects window we can see all the generics, ports and signals in the entity that is activated in the Sim tab. In *Figure 18* the top-level design is activated so the signals in the top-level design are visible. We can see that we have three 4-bit vector signals given on hexadecimal format. The signals all have unknown values X since we haven't started the simulation yet.

If we click on one of the plus signs  by one of the 4-bit signals, here the `a_tb_signal`, *Figure 20*, we will see the individual bits within the current vector

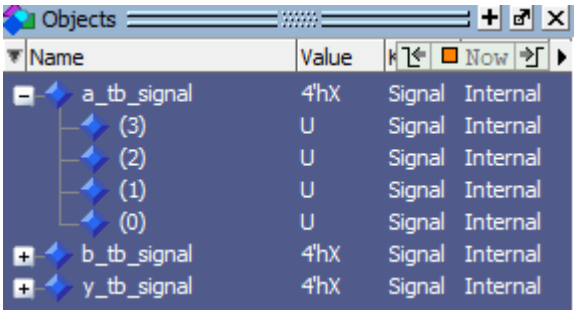


Figure 20 Objects window with opened vector

signal. The letter U indicates that the bits haven´t been assigned any values yet. You can go back to just seeing the vector by clicking on the menu´s sign  that have replaced the plus sign.

If we activate one of the components in the sim window the Objects window will change and the signals within that component will be visible, *Figure 21*. You cannot see variables within processes this way, but we shall see that there are other ways to make them visible.

In *Figure 21* we can see the signals in the component `four_bit_full_adder` that the test bench is using but is we go down one level more we will not see any signals within the `full_adder` component since they have been optimized away. If the do the simulation without optimization they will be visible though and in *Figure 22* we see the signals of one of the 1-bit full adders.

There is also a Wave window where we will see the resulting waveforms from the simulation, *Figure 23*.

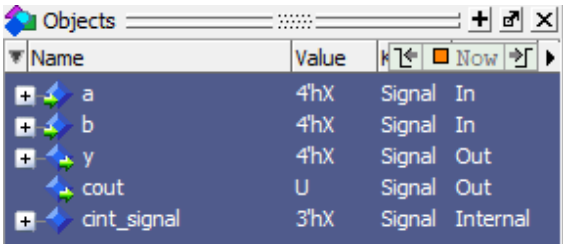


Figure 21 Objects window for the four bit ripple adder component

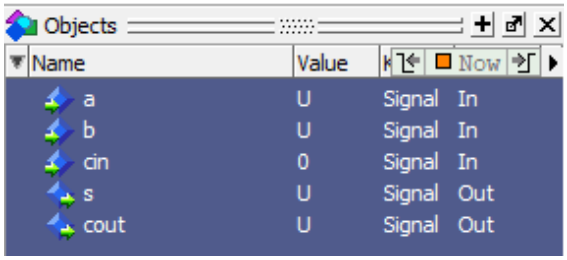


Figure 22 Objects window for a component

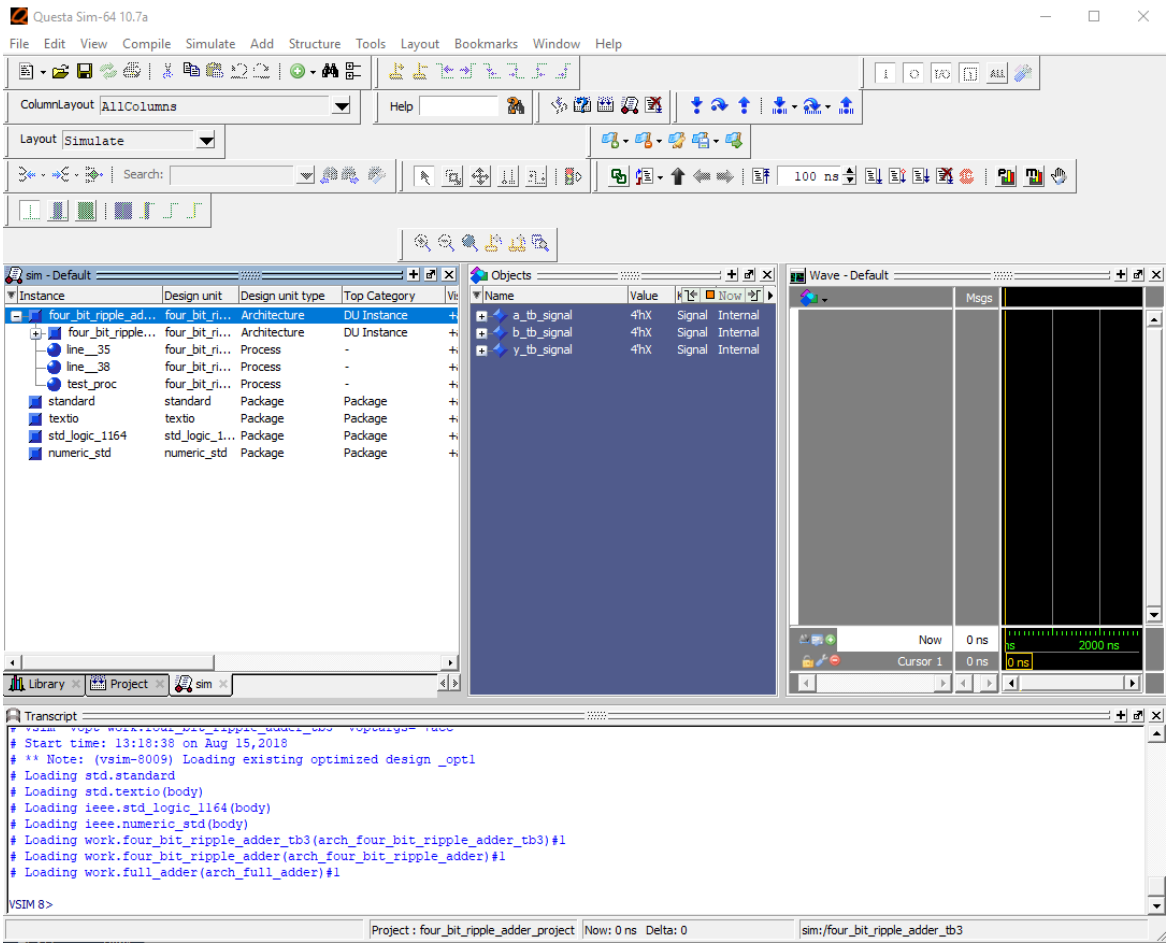


Figure 23 GUI in simulation mode with Wave window

If the Wave window isn´t open, you can open it by the menu choice

View -> Wave

If you haven´t opened the Wave window it will automatically show when you choose to add items to it. You add a signal to the Wave window by right clicking on the signal in the Objects window and select Add Wave.
You can also type

```
add wave <signal_1_name> [<signal_2_name>]
```

In the Transcript window. Notice that when you add more than one signal the signal names should be separated by a space.

You select several of the signals to be shown in the Wave window by using the normal Windows way of selecting more than one object in the Objects window and then right click and select

Add -> To Wave -> Selected Signals

You can also drag the selected signals from the Objects window to the Wave window. You can choose

Add -> To Wave -> Signals in Region

instead which will add all signals at the current level of hierarchy to the Waveform window. You can also select

Add -> To Wave -> Signals in Design

This will not only add the signals in the region but also signals in sub designs like components to the Wave window. The names of the signals from sub designs show up like

<name_of_sub_circuit>/<signal_name>

To see variables within processes you need to add them by using commands in the Transcript window or using do files (more on this later). In the Transcript window or in the do file you write

```
add wave <name_of_process>/<variable_name>
```

As you can see you need to name the process to make this work. For this to work you must also have started your simulation without optimization or have changed No design object visibility to Apply full visibility to all modules (full debug mode) in the Optimization Options window, *Figure 16*.
As you can see in *Figure 23* the Wave window in the GUI is quite small and of little use as it is. Unlock the window by clicking on the icon  in the right top corner of the window to turn it into a floating window and drag out the window to a useful size.
In the Waveform window, *Figure 24*, the top-level signals and ports and the signals and ports from the full_adder comp_1 are added. The window will have three regions.

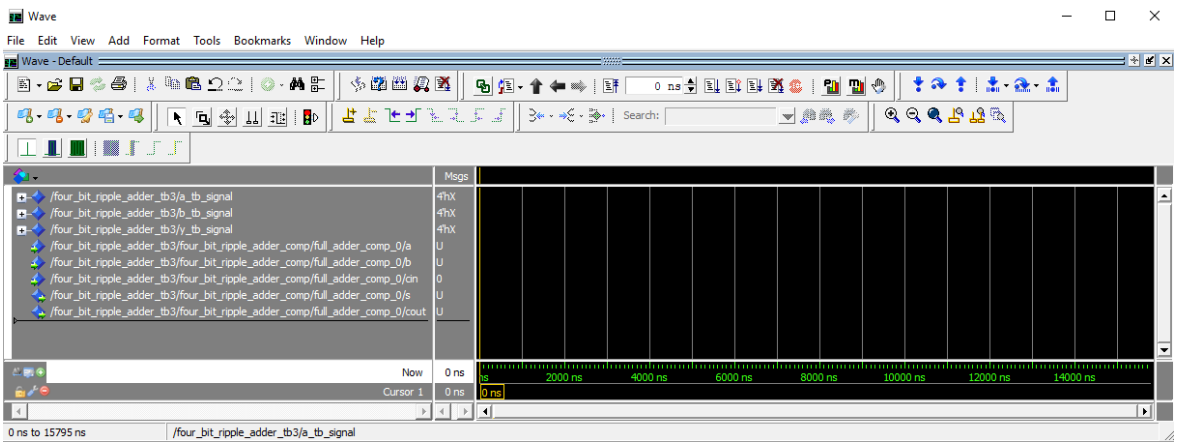


Figure 24 Waveform window

- A list of the values of the objects at the time given by the position of the cursor line in the Wave display
- You can change the radix of the signals. You can for example watch the value of a vector in binary, hexadecimal, decimal (signed) or unsigned form. The choice of radix can be done on each signal individually. You change the radix by right clicking on the signal name or the signal value and selecting Radix and then select the radix you want. By selecting more than one signal before you change the radix you can change their radixes all at once. You can also set the radix from the do file. You can also add the same signal several times and then select different radix in the instances.
- A waveform area showing the waveforms graphically

The width of these regions can be changed by grabbing the separators between the regions with the mouse and dragging them. This might be necessary to be able to see the full signal names.

Before you start running simulation time all ports and signals will have undefined values. They will not take on any default values, so you have to make sure to give them appropriate values. This is different from the real synthesized devices where the signals always will have some kind of value. Any objects that are given values in their declaration will take on these values but these values will only work in simulation and not in hardware. Since this means that the behavior is different in simulation and synthesis it is not wise to set default values to signals. Instead use some kind of reset phase to set the start values for your simulation. By doing that you will also get start up values for the synthesized design. U indicates unassigned inputs while X indicates undetermined values on outputs. The X might be there because it is an output signal that hasn´t got any value yet since we haven´t run any simulation time. The cause of the X can also be that the signal is connected to two different terminals that try to write different values to the node.

You can give all our simulation commands in the Transcript window although putting them in a script file, a do file, is a better choice since by doing that it´s simple to repeat exactly the same simulation when you have done some editing of the source code.

Since the do file is a script file it´s just a text file that contain a list giving exactly the same commands as those you can input from the Transcript window with one command per line.

Simulation commands

You give the signals values by using the command

```
force <signal_name> <value>
```

For binary scalars, you can force the value to the allowed values for the data type. You have for example nine (9) choices for a `std_logic` scalar. The command

```
force a Z
```

will set the signal to a high impedance level Z.

For vectors, you give values to all the bits using for example

```
force b 4'b0110
```

This command says that we are giving the vector `b` a 4-bit (4) binary value (b) `0110`. To use decimal, octal or hexadecimal values we use `d`, `o` or `h` as type definers instead, still giving the number of bits as a value. If the value is too big for the given number of bits, the MSB´s will be removed from the value.

```
force b 4'h6
```

and

```
force b 4'd6
```

will give the same result as the first binary command.

To set a negative number you don´t have to get the value using 2's complement. To set the value -6 you can just write

```
force b -4'b0110
```

```
force -4'h6
```

or

```
force -4'd6
```

The first digit, here 4, that should give the number of bits doesn´t seem to matter. The value will adopt to the format of the signal the value should control. If the number of values is bigger than this digit, then the last values will be truncated.

In earlier versions of QuestaSim binary representation was default and you could give a binary value as

```
force a 1110
```
Incorrect

but that is no longer valid.

You can also give the time when the signal should get the new value in the `force` statement.

The time can be given as just a number

```
force a 0 100
```

where the time is given in the default time base of the simulator which is nanoseconds (ns). The command can also be given with a time base that might not be the default one

```
force a 0 100ns
```

The latter form might be somewhat clearer. Both commands mean that the signal a should be set to zero (0) at time 100 ns counting from the current simulation time.
With the command

```
force a 0 @100
```

you will set the signal a to zero (0) at the absolute simulation time 100 ns from the start of the simulation instead of at 100 ns from the current time. You can only use this command to set values at times that haven´t passed in the simulation yet.

You can also give a sequence of values at different times in one command. The command

```
force a 0 0,1 100us,0 150us
```

means that the signal a should be set to zero (0) at current time, be set to one (1) 100 µs (microseconds) later and then be set to zero (0) again at 150 µs from current time. The ordering of the times must be consecutive.

You can create a repeating signal, for example a clock signal, with the command

```
force a 0 0,1 50ns –repeat 100ns
```

which means that the signal a is set to zero (0) at the current time, set to one (1) 50 ns later and then a time period of 100 ns is repeated infinitely. We have created a symmetrical clock signal that starts with a zero (0) and has a period of 100 ns.

The `force` command will not run the simulation, it will only set the values, and nothing will happen until you run some simulation time. Remember to set all signals before you start the simulation so that none of them are undefined. An undefined signal in the Waveform window will be indicated in red.

You can give more than one `force` command, one after the other, and they will all be effective at the same time as long as you don´t run any simulation time in between. If you run some simulation time between two `force` commands, then the latter command will take effect after that simulation time. Avoid changing more than one signal at the same time though since there may be some confusion in the timing of the signals. It´s only in simulation that signals can change at exactly the same time, in reality they don´t.

After this you will have to set the time that you want the simulation to run. You do this with the command

```
run <time_to_simulate>
```

A `force` command that is not followed by a run command will be meaningless since no simulation time will be run.

You can find an example of the Waveform window for a simulation in *Figure 25* where the test bench for a 4-bit adder is simulated for a number of input stimuli using a test bench. The + signs by the names of the vectors indicates that the display can be expanded to show the individual bits of the vector. This has been done for vector a and here you see a − sign to shrink it back to just a vector.

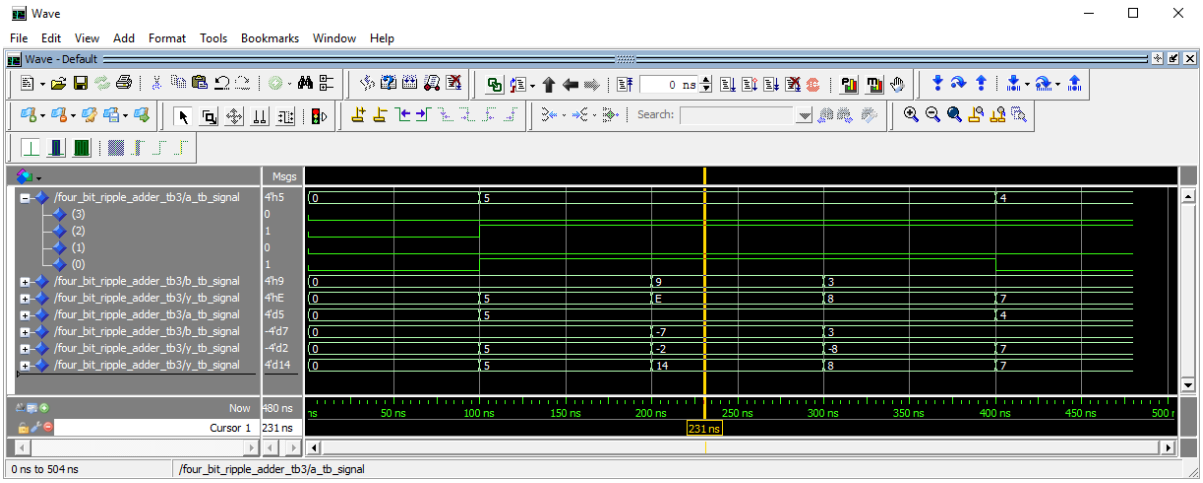


Figure 25 Waveform window with simulation result

When you left click in the waveform a cursor will show at that position accompanied by the time of that cursor position and the values at that time are shown in the Msgs column.
In *Figure 25* the 4-bit input signals a and b are shown twice, once in hexadecimal form and once in decimal (unsigned) form. The output vector y is shown three times, in hexadecimal form, in signed decimal form and in unsigned decimal form. There is a cursor at time 231 ns and the signal values at that time are given in the Msgs column.

In the Wave window, you can do zooming operations. You have a number of zooming options in the

View -> Zoom

menu. You can zoom in and out, zoom in around the cursor, zoom the entire simulation time or zoom a selected region. There are also some quick buttons for zooming  where you can zoom in  and out  , zoom the entire simulation time  , zoom in around the cursor  and zoom between two cursors  .

You can add more cursors to the display by right clicking on the time scale in the waveform window and select

New Cursor @ <time>

Where <time> is the position of your mouse pointer.

When you have more than one cursor you will not only be able to see the times of the cursors but also the time differences between the cursors.

The simulation can be rolled back to time zero with the command

```
Restart
```

Automating the simulation, do files

When you simulate you can give the stimuli and run the simulation interactively, but this will be tiresome and error prone and if you want to rerun the same simulation sequence, something you most likely want to do after error correction and/or redesign of your construction, you have to remember and type in the sequence to run. Things are a bit simplified by the fact that you can navigate among the given commands using the up and down keys on the keyboard, there is a history list for the commands.

To make things even simpler you can run the entire simulation using a script file, a do file in QuestaSim language.

To do this you create a text file with the commands that you want to run, one command per line in the order you want them to be executed.

Normally the do file has the ending `.do` but this is not default which means that you cannot leave out the ending when you type the name of the do file that you want to run. The file can actually have any ending but it's good practice to let it be identified by its `.do` ending.

You can add the do file to the project. This has no significant meaning to the design, but it is a way of keeping track of the files belonging to the project and it means that you can easily open the file by double clicking on it in the Project window and you don´t have to use a file browser to open the file. If the do file is added to the project, it can also be run by right clicking on it in the Project tab and selecting Execute. By doing it this way the project will keep track of in what folder the do file is placed and you don´t have to give any search path although the file might not be placed in the project folder.

You run the do file from the Transcript window by typing

```
do <file_name.do>
```

don´t forget the file ending. This will only work if the do file is placed in the project folder, if it´s not, you will have to include the search path. If you type the first letter(s) in the name of the do file and then hit tab the system will try to match what you have typed to the file name and fill in the rest of the name. This will only work as long as the string of letters is unique and doesn´t match more than one file. If the string matches more than one file these files will be shown below the Transcript window and you can use the mouse to select the file you want to use.

You can include the setting up of the Wave window and the addition of the signals you want to see in the do file. You can even write a do file that does the whole flow; compiles the

design files, starts the simulator and runs the simulation but in most cases, you don´t gain much from that since you need to check for errors in the design before you move from compilation to simulation.

You start the do file by restarting the simulation and removing all the signals from the Wave window with the command

```
restart –f –nowave
```

This options in the command will have no effect the first time you run the simulation, but it will restart and clean things up the next time you run it.

After this you should give the same commands as the ones you earlier typed in the Transcript window to make them run directly but now you place the commands in the do file in the order you want them to run instead of manually giving them as a sequence of commands in the Transcript window.

You start by declaring the windows you want to see. The command

```
view signals wave
```

means that you want to see the Objects (signals) and Wave windows.

Then you add the signals that you would like to see in the Wave window with the command

```
add wave <space_separated_list_of_signals>
```

Notice that the list is space separated, there are no commas or semicolons. For example

```
add wave clock a b
```

will add the signals clock, a and b to the Wave window.

You can set the radix of the displayed signals by including it in the command

```
add wave clk count –radix unsigned count
```

Here we add the signal count twice, once with the default radix hexadecimal and once with the radix unsigned decimal. You have the same selection of radixes as we described in the Wave window earlier.

The radix command will stick for the rest of the line, so you have to change it if you want another radix for coming signals. Another way of going back to the default radix is to use a new add wave command for the rest of the signals.

You can also watch variables although they are local to a process but then you have to tell the simulator in which process they are located and this means that this process will have to have a name. If you have a variable test_OK in the process named test_process you would write

```
add wave test_process/test_OK
```

to see the variable `test_OK` in the Wave window.
The same kind of command can be used to see signals inside components that are instantiated in the design, you write

```
add wave name_of_instantiated_component/name_of signal
```

As we described earlier this will not always work though since the signals might be optimized away. You will then have to start the simulation with full visibility as described earlier.

Comments can be added to the do file. The comments should be placed on separate lines, not on lines with code and the comment should start with a # sign.

Example

Let´s take an example. To get just a small code section we will use a 1-bit full adder.

We start by creating a new project in some folder. We call the project `full_adder`. At the same time we create a new design file called `full_adder.vhdl`.

We write our VHDL code in the design file

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY full_adder IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         cin:IN STD_LOGIC;
         s:OUT STD_LOGIC;
         cout:OUT STD_LOGIC);
END full_adder;

ARCHITECTURE arch_full_adder OF full_adder IS
BEGIN
   s<=(a XOR b) XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

and compile the code. If we have written the code correctly the compilation should be successful.

Let us move to simulation. We start by creating the simulation script, the do file, and call this file `full_adder.do`.

To test all possible signal values, we have to use eight different input stimuli, *Figure 24*.

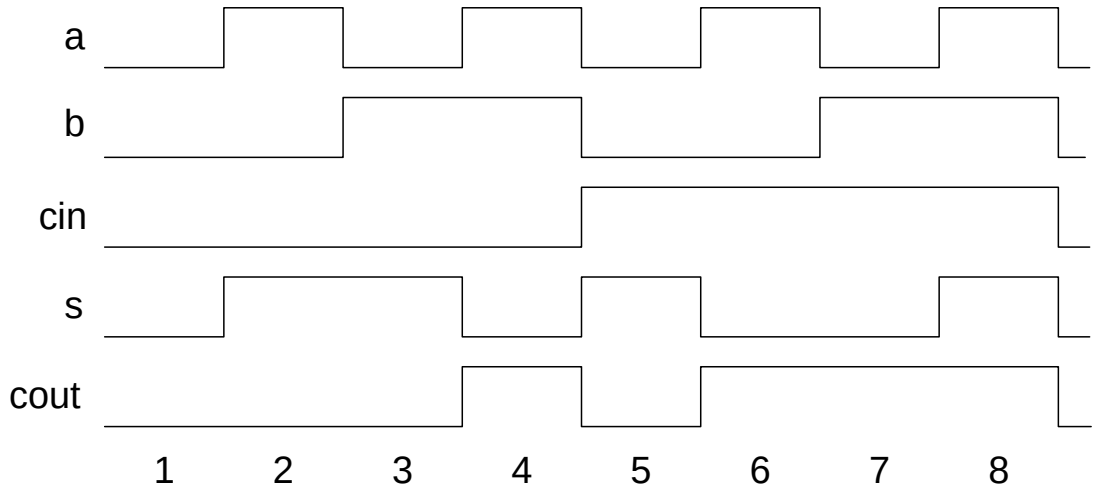


Figure 24 Input and output waveforms for the full adder

We want to see all signals in the simulation and to make the simulation thorough we go through all signal combinations. Let´s say that we let 100 ns pass between each signal change. In the interest of not complicating things at the moment we will diverge from the recommendation to only let one signal change value at any given time. We write the do file

```
restart –f –nowave
view signals wave
add wave a b cin s cout
# time 0 ns 000
force a 0
force b 0
force cin 0
run 100ns
# time 100 ns 100
force a 1
run 100ns
# time 200 ns 010
force a 0
force b 1
run 100ns
# time 300 ns 110
force a 1
run 100ns
# time 400 ns 001
force a 0
force b 0
force cin 1
run 100ns
```

```
# time 500 ns 101
force a 1
run 100ns
# time 600 ns 011
force a 0
force b 1
run 100ns
# time 700 ns 111
force a 1
run 100ns
```

We could rewrite the file by assigning values signal by signal

```
restart –f –nowave
view signals wave
add wave a b cin s cout
force a 0 0ns, 1 100ns,0 200ns,1 300ns,0 400ns,
                                    1 500ns,0 600ns,1 700ns
force b 0 0ns,1 200ns,0 400ns,1 600ns
force cin 0 0ns,1 400ns
run 800ns
```

The line split in the `force a` command is editorial, it cannot be there in the `do` file. It can be split into two `force` commands though. You will either have to keep it on one line or split it into two `force` commands.
This way of giving stimuli gets quite tiresome and a bit hard to keep track of. If you study the three input signals you can see that they behave as three repeating clock signals with periods of 200 ns, 400 ns and 800 ns and they all start with the value zero (0). This means that you could write the `do` file as

```
restart –f –nowave
view signals wave
add wave a b cin s cout
force a 0 0ns, 1 100ns –repeat 200ns
force b 0 0ns,1 200ns –repeat 400ns
force cin 0 0ns,1 400ns –repeat 800ns
run 800ns
```

In the Wave window, you can see the simulated signals zoomed out to a full window, *Figure 25*. The cursor is placed at time 351,489 ns and in the values region, you can see what values that the signals have at that time.

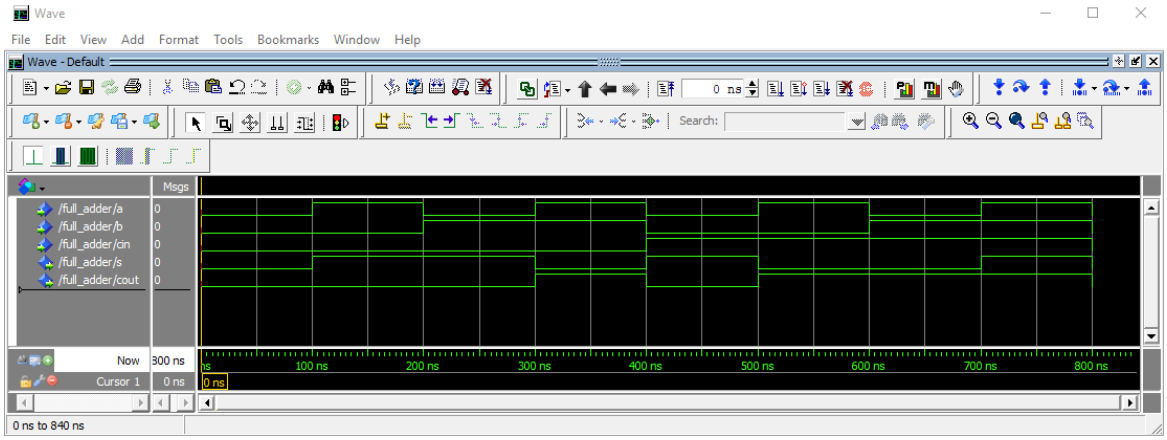


Figure 25: Simulated waveforms for a full adder

Testbenches

We have looked at running a simulation in QuestaSim but so far we have only generated input stimuli and we evaluate the results by looking at the resulting waveforms. What if we could also write code to test the results from the simulation? This is what Testbenches are all about.

Instead of directly simulating your design you use it as a component in a system level design, a testbench. This top-level design generates input stimuli to the component, your design, and can also read the resulting outputs and check if they are valid, *Figure 26*.

There are three types of testbenches. They don´t really have any names but I call them type 1, 2 and 3.

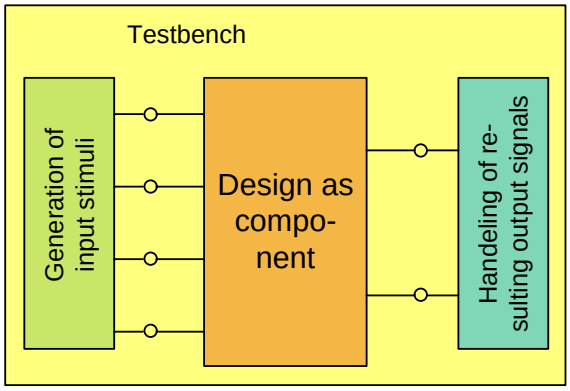


Figure 26 Testbench structure

- **Type 1:** The testbench only generates input stimuli and it´s up to the user to check the resulting output signals. This means that it behaves just like a `do` file
- **Type 2:** The testbench generates input stimuli and checks the results. It uses a test output, a flag often called `test_OK_signal`, to signal if the result is correct. The value is held high (1) as long as the results are correct. As soon as the result is incorrect the flag will go low (0) indicating the error and it will stay low from that on although the results later in the simulation might be correct
- **Type 3:** The testbench generates input stimuli and checks the results. Instead of using a flag it will issue a message when an incorrect result occurs. It will also indicate at what time in the simulation run the incorrect result occurred. It´s up to the designer to formulate the messages so it can be quite extensive. We can decide either to stop the simulation when an incorrect result occurs or continue with the simulation. In the latter case, more messages will be presented if new incorrect results occur

Example

Let´s write all three kinds of testbenches for the earlier ripple carry adder design.
In all three designs, we will us our full adder as a component. The full adder had the entity

```
ENTITY full_adder IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         cin:IN STD_LOGIC;
         s:OUT STD_LOGIC;
         cout:OUT STD_LOGIC);
END full_adder;
```

The basic testbench design is the same in all three cases. The difference is if and how we check the simulation results.
The entity for all testbenches will be the same and the architecture for the testbench of type 1 will be part of the architectures for the testbench of type 2 and 3 so we start with the type 1 testbench.

Testbench type 1

First the entity of the testbench

```
ENTITY full_adder_tb1 IS

END full_adder_tb1;
```

Notice that the entity is empty, we don´t have any in- or outputs, we only have internal signals.
We create an architecture where we instantiate our full adder as a component and declare signals that we connect to the inputs and outputs of the full adder.

```
ARCHITECTURE arch_full_adder_tb1 OF
                          full_adder_tb1 IS

   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC;
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;

   SIGNAL a_tb_signal:STD_LOGIC;
   SIGNAL b_tb_signal:STD_LOGIC;
   SIGNAL cin_tb_signal:STD_LOGIC;
   SIGNAL s_tb_signal:STD_LOGIC;
   SIGNAL cout_tb_signal:STD_LOGIC;
```

```
BEGIN
   full_adder_comp:
   COMPONENT full_adder
      PORT MAP(a=>a_tb_signal,
               b=>b_tb_signal,
               cin=>cin_tb_signal,
               s=>s_tb_signal,
               cout=>cout_tb_signal);
```

To complete the basic architecture, we assign values to the signals connected to the inputs of the component. Notice that we can´t assign values directly to the inputs of the component but we must use signals for this. We generate the same signal pattern as we had in *Figure 24*.

```
   a_tb_signal<='0',
               '1' AFTER 100 ns,
               '0' AFTER 200 ns,
               '1' AFTER 300 ns,
               '0' AFTER 400 ns,
               '1' AFTER 500 ns,
               '0' AFTER 600 ns,
               '1' AFTER 700 ns;
   b_tb_signal<='0',
               '1' AFTER 200 ns,
               '0' AFTER 400 ns,
               '1' AFTER 600 ns;

   cin_tb_signal<='0',
               '1' AFTER 400 ns;
```

The first signal assignment without a time stamp sets the signal value at time zero (0) while the following assignments sets the value at the given times.
In general, we don´t use time in VHDL code since hardware can´t handle that but we can use it here since the testbench is just for simulation and will never be placed in hardware.
We can simplify the assignment a bit in a similar way to how we used clock signals in the earlier do file.

```
   a_tb_signal_process:
   PROCESS
   BEGIN
      WAIT FOR 100 ns;
      a_tb_signal<=NOT(a_tb_signal);
   END PROCESS a_tb_signal_process;
```

```
b_tb_signal_process:
PROCESS
BEGIN
   WAIT FOR 200 ns;
   b_tb_signal<=NOT(b_tb_signal);
END PROCESS b_tb_signal_process;

cin_tb_signal_process:
PROCESS
BEGIN
   WAIT FOR 400 ns;
   cin_tb_signal<=NOT(cin_tb_signal);
END PROCESS cin_tb_signal_process;
```

For this to work we must assign default values, start values, to our input signals otherwise the simulator don´t know what values to invert. We change the signal declarations to include these.

```
SIGNAL a_tb_signal:STD_LOGIC:='0';
SIGNAL b_tb_signal:STD_LOGIC:='0';
SIGNAL cin_tb_signal:STD_LOGIC:='0';
```

Once again this is something we shouldn´t do when we write code for hardware. Default values will not transfer to hardware. For this to work in hardware we need some kind of reset phase.
We still need a do file where we can declare what signals we want to see in the Wave window and to run some simulation time.

```
restart -f -nowave
view signals wave
add wave a_tb_signal b_tb_signal cin_tb_signal
add wave s_tb_signal cout_tb_signal
run 780ns
```

Here we have two `add wave` commands just to shorten the command lines.
When you run the simulation, you might need to simulate without optimization. The reason for this is that since we only have signals and no output ports the simulator assumes that these signals can be optimized away as they don´t connect to any ports.

Testbench type 2

Here we keep the testbench type 1 but expand it to include checking the resulting output signals.
First we add a signal to indicate the test result

```
SIGNAL test_OK_signal:STD_LOGIC;
```

Instead of a signal we can define `test_OK` as an out port if we like. The result will be the same.
Then we add a process for the test. Here the process is shortened so it doesn´t take up that much space.

```
    test_proc:
    PROCESS
    BEGIN
       test_OK_signal<='1';
       WAIT FOR 50 ns; -- 50 ns 0+0+0
       IF s_tb_signal/='0' OR cout_tb_signal/='0' THEN
          test_OK_signal<='0';
       END IF;
       WAIT FOR 100 ns; -- 150 ns 1+0+0
       IF s_tb_signal/='1' OR cout_tb_signal/='0' THEN
          test_OK_signal<='0';
       END IF;

--        Five wait statements have been removed to shorten
--        the text

       WAIT FOR 100 ns; -- 750 ns 1+1+1
       IF s_tb_signal/='1' OR cout_tb_signal/='1' THEN
          test_OK_signal<='0';
       END IF;
    END PROCESS test_proc;
```

Notice that the signal `test_OK_signal` is assigned the default value one (1) when we start the process and then will be set low (0) as soon as an incorrect signal occurs, and the signal will never be set to one (1) again unless we run for so long time that the process starts all over again.
We use the same `do` file as for testbench type 1 with the difference that we add `test_OK_signal` to the signals we want to see in the Wave window.

Testbench type 3

The only thing that really change for the testbench type 3 is the test process, but we can take away the `test_OK_signal` since this will not be used any more. Once again the process is shortened.

```
    test_proc:
    PROCESS
    BEGIN
       WAIT FOR 50 ns; -- 50 ns 0+0+0
       ASSERT (s_tb_signal='0' AND cout_tb_signal='0')
       REPORT "Error for 0+0+0"
       SEVERITY ERROR;
```

```
        WAIT FOR 100 ns; -- 150 ns 1+0+0
        ASSERT (s_tb_signal='1' AND cout_tb_signal='0')
        REPORT "Error for 1+0+0"
        SEVERITY ERROR;

--        Five wait statements have been removed to shorten
--        the text

        WAIT FOR 100 ns; -- 750 ns 1+1+1
        ASSERT (s_tb_signal='1' AND cout_tb_signal='1')
        REPORT "Error for 1+1+1"
        SEVERITY ERROR;
    END PROCESS test_proc;
```

We can realize that this file can be quite long if we have many things to test.
In the process the ASSERT statement indicates what values that should be true in the test and if they are then no message will be issued.
REPORT gives the message that is issued if the assertment is false. In the example, it´s a simple text message but it can be more complicated including signal values and other things in the message. Beside the REPORT message the time when the message is asserted will also be written to the Transcript window.
SEVERITY indicates what should happen at an incorrect output signal. SEVERITY can have four different values that will be given different headers

- NOTE, this is only a comment it´s no real error
- WARNING¸ indicates something to look out for but it´s not an error
- ERROR, used when an error is detected but the simulation will not be stopped
- FAILUE, used when an error is detected and is so severe that the simulation is stopped

It´s up to the designer to decide what should be looked upon as NOTE, WARNING or ERROR since the behavior is the same in all three cases.